Vince

ACCENT QNIX MANUAL

VOLUME TWO

Preliminary Draft for Review

June 18, 1985

Reviewers:

Nick Felisiak, Spider
Ellen Colwell
Bob Mitchell
Vicki Preston
Circulating copy
(Joyce Swaney, Don Scelza)

See instructions and notes on cover of Volume One.

ACCENT QNIX MANUAL

VOLUME ONE

VOLUME TWO

NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls.

Most of the system calls have one or more possible error conditions. An error condition is indicated by an otherwise impossible returned value. This value is almost always −1; each individual call description specifies the actual value. The number of the error is made available in the external variable *errno*. *Errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

Each system call description attempts to list all possible error conditions. The following is a complete list of the error numbers and their names as defined in **<errno.h>**.

1 EPERM Not owner
: An attempt was made to modify a file in some way forbidden except to its owner. This is also returned for attempts to do things that are privileged facilities not supported in QNIX.

2 ENOENT No such file or directory
: A specified file does not exist, or one of the directories in a pathname does not exist.

3 ESRCH No such process
: No process can be found corresponding to the one specified by *pid* in *kill*.

4 EINTR Interrupted system call
: An asynchronous signal (such as interrupt or quit) that the user has elected to catch occurred during a system call. If execution is resumed after processing the signal, it appears as if the interrupted system call returned this error condition.

5 EIO I/O error
: Some physical I/O error has occurred. This error sometimes occurs on a call following the one to which it actually applies.

6 ENXIO No such device or address
: I/O on a special file refers to a subdevice that does not exist or refers beyond the limits of the device.

7 E2BIG Arg list too long
An argument list longer than 5,120 bytes was presented to a member of the *exec* family.

8 ENOEXEC Exec format error
A request was made to execute a file that is not an executable file, although it has the appropriate permissions.

9 EBADF Bad file number
Either a file descriptor refers to no open file, a read request is made to a file that is open only for writing, or a write request is made to a file that is open only for reading.

10 ECHILD No child processes
A *wait* was executed by a process that had no existing or unwaited-for child processes.

11 EAGAIN No more processes
A *fork* failed because the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough space
During an *exec*, *brk*, or *sbrk*, a program asks for more space than the system is able to supply. This is not a temporary condition; the maximum space available is a system parameter.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
The system encountered an address translation fault in attempting to use an argument of a system call.

17 EEXIST File exists
An existing file was mentioned in an inappropriate context.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required (for example, in a path prefix or as an argument to *chdir*).

21 EISDIR Is a directory
An attempt was made to write on a directory.

22 EINVAL Invalid argument
An invalid argument was given (e.g., mentioning an undefined signal in *signal* or *kill*, or reading or writing a file for which *lseek*

has generated a negative pointer).

23 ENFILE File table overflow
The system file table is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
No process may have more than 20 file descriptors open at a time.

27 EFBIG File too large
The size of a file exceeded the maximum file size (1,082,201,088 bytes) or ULIMIT.

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An *lseek* was issued to a pipe.

32 EPIPE Broken pipe
A write was attempted on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (3M) is not representable within machine precision.

DEFINITIONS

Process ID
Each active process in the system is uniquely identified by a positive integer called a process ID.

Parent Process ID
A new process is created by a currently active process; see *fork*. The parent process ID of a process is the process ID of its creator.

Process Group ID
Each active process is a member of a process group that is identified by a positive integer called the process group ID. This grouping permits the signaling of related processes; see *kill*.

Real User ID

Each user allowed on the system is identified by a positive integer called a real user ID. The real user ID of an active process is the real user ID of the user who created the process.

File Descriptor

A file descriptor is a small integer used to do I/O on a file. The value of a file descriptor ranges from 0 to 19. A process may have no more than 20 file descriptors (0-19) open simultaneously. A file descriptor is returned by system calls such as *open* or *pipe*. The file descriptor is used as an argument by calls such as *read*, *write*, *ioctl*, and *close*.

Filename

Strings of 1 to 80 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

Note that it is generally unwise to use *, ?, :, [, or] as part of filenames because of the special meaning attached to these characters by the shell; see *sh*(1). Although permitted, using unprintable characters in filenames should be avoided.

Pathname and Path Prefix

A pathname is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a filename. An extended pathname (a searchlist name prepended to a pathname) may also be used. Searchlist names are terminated by a colon.

More precisely, a pathname is a null-terminated character string constructed as follows:

<pathname>::=<filename>|<pathprefix><filename>|/
<pathprefix>::=<rtprefix>|/<rtprefix>|<searchlist><rtprefix>
<rtprefix>::=<dirname>/|<rtprefix><dirname>/

where <filename> and <dirname> are strings of 1 to 80 characters permitted in names, and <searchlist> is a string of 1 to 80 characters permitted in names terminated by a colon (see **Filename** above).

If a pathname begins with a searchlist, the start of the path search iterates over each directory in the searchlist for a read-only action. For actions that modify a file (write to it or delete it), only the first element of the searchlist is examined.

If the pathname does not begin with either a slash or a searchlist, the search begins from the current working directory (the *current:* searchlist).

A slash by itself names the root directory.

QNIX allows path prefixes to be made 'well-known'. When a reference is made to a filename starting with a well-known prefix, an additional component is prepended to the name. This allows commonly used UNIX directory names (e.g. */bin*) to be used in a QNIX environment. Setting up and using well-known prefixes is explained in the *QNIX User Guide.*

If a pathname begins with a slash, the path search begins at the *root* directory.

Unless specifically stated otherwise, the null pathname is treated as if it named a non-existent file.

Directory

Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory

A root directory and a current working directory is associated with each process for the purpose of resolving pathname searches. The root directory of a process need not be the root directory of the root file system.

File Access Permissions

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The user ID of the process matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The user ID of the process does not match the user ID of the owner of the file, but the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied.

SEE ALSO

close(2), open(2), pipe(2), read(2), write(2), intro(3).

NAME

access – determine accessibility of a file

SYNOPSIS

```
int access (path, amode)
char *path;
int amode;
```

DESCRIPTION

Access checks the file named by the pathname pointed to by *path* for accessibility according to the bit pattern in *amode*. *Amode* is interpreted as follows:

04	read
02	write
01	execute (search)
00	check existence of file

Access to the file is denied if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EACCES]	Permission bits of the file mode do not permit the requested access.
[EFAULT]	*Path* points outside the allocated address space for the process.

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits. All others have permission checked with respect to the "other" mode bits.

RETURN VALUE

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

QNIX

At release 1 of QNIX, *access* checks only for the existence of the file. Permissions are not checked.

SEE ALSO

chmod(2), stat(2).

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

```
int brk (endds)
char *endds;

char *sbrk (incr)
int incr;
```

DESCRIPTION

Brk and *sbrk* dynamically change the amount of space allocated for the calling process's data segment (see *exec*(2)). They reset the calling process's break value and allocate the appropriate amount of space. The break value is a pointer in a block of virtual memory managed by these calls. Newly allocated space is set to zero.

Brk sets the break value to *endds* and changes the allocated space accordingly.

Sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased.

Brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

The requested change would result in more space being allocated than is allowed by a system-imposed maximum. [ENOMEM]

The requested change would result in the break value being greater than or equal to the start address of any attached shared memory segment.

RETURN VALUE

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

ACCENT MEMORY MANAGEMENT

Under QNIX, the data segment allocated by calls to *brk* and *sbrk* is not contiguous with the compiled data. A region of 4Mb is allocated on the first call to *brk* or *sbrk*, and the calls simply manage the use of this area. Additional virtual memory may be allocated using Accent memory management calls. If the Accent calls are used, care should be taken that specific addresses are not given, since doing so may interfere with the QNIX mechanisms.

SEE ALSO

exec(2), malloc(3C), malloc(3X).
The ValidateMemory section of the "Kernel Interface" document in the *Accent Programming Manual*.

NAME

chdir – change working directory

SYNOPSIS

int chdir (path)
char *path;

DESCRIPTION

Chdir causes the directory named by *path* to become the current working directory (the starting point of searches for path names not beginning with /).

Chdir will fail and the current working directory will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*Path* points outside the allocated address space of the process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

chmod – change mode of file

SYNOPSIS

int chmod (path, mode)
char *path;
int mode;

DESCRIPTION

Chmod sets the access permission portion of the mode of a file according to the bit pattern contained in *mode*. The file is named by the pathname pointed to by *path*.

Access permission bits are interpreted as follows:

00400	Read by owner.
00200	Write by owner.
00100	Execute (search if a directory) by owner.
00007	Read, write, execute (search) by others.

The user ID of the process must match the owner of the file to change the mode of a file.

Chmod will fail and the file mode will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The user ID does not match the owner of the file.
[EFAULT]	*Path* points outside the allocated address space of the process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

mknod(2).

NAME

close – close a file descriptor

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

Close closes the file descriptor indicated by *fildes*. *Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Close will fail if:

[EBADF] *Fildes* is not a valid open file descriptor.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

int creat (path, mode)
char *path;
int mode;

DESCRIPTION

Creat creates a new ordinary file or prepares to rewrite an existing file. The file is named by the pathname pointed to by *path*.

If the file already exists, its length is truncated to 0 and its mode and owner are unchanged.

If the file does not exist, the new file's owner ID is set to the user ID of the process, and the low-order 9 bits of the file mode are set to the value of *mode*. The mode is modified so that all bits set in the process's file mode creation mask are cleared (see *umask*(2)).

Upon successful completion, the file descriptor is returned and the file is open for writing, even if its mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls (see *fcntl*(2)). No process may have more than 20 files open simultaneously. A new file may be created with a mode that forbids writing.

Creat will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[ENOENT]	The pathname is null.
[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.
[EACCES]	The file exists and write permission is denied.
[EISDIR]	The named file is an existing directory.
[EMFILE]	Twenty file descriptors are currently open.
[EFAULT]	*Path* points outside the allocated address space of the process.

[ENFILE] The system file table is full.

RETURN VALUE

Upon successful completion, a non-negative integer (the file descriptor) is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), open(2), read(2), umask(2), write(2).

NAME

dup – duplicate an open file descriptor

SYNOPSIS

int dup (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

The new file descriptor is set to remain open across *exec* system calls (see *fcntl*(2)).

The file descriptor returned is the lowest one available.

Dup will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EMFILE] Twenty file descriptors are currently open.

RETURN VALUE

Upon successful completion, a non-negative integer (the file descriptor) is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

execl, execv – execute a file

SYNOPSIS

int execl (path, arg0, arg1, ..., argn, 0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

DESCRIPTION

Exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary executable file called the *new process file*. This file consists of a header, a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss).

There can be no return from a successful *exec* because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

```
main (argc, argv, envp)
int argc;
char **argv;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. By convention, *argc* is at least one and the first member of the array points to a string containing the name of the file.

Arguments to *exec* commands are as follows:

Path points to a pathname that identifies the new process file.

File points to the new process file. The path prefix for this file is obtained by a search of the directories in the *run:* environment searchlist variable.

Arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list

available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

The UNIX *environment* is implemented differently in QNIX. The *getenv*(3C) library routine will access Accent environment strings in the ordinary way. Environment searchlists cannot be accessed by UNIX interfaces, but Accent Environment Manager routines may be used. Use of the *environ* external variable is not supported.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set (see *fcntl*(2)). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate the new process. See *signal*(2).

The new process also inherits the following attributes from the calling process:

process ID

parent process ID

process group ID

current working directory

root directory

file mode creation mask (see *umask*(2))

Exec will fail and return to the calling process if one or more of the following are true:

[ENOENT] One or more components of the pathname of the new process file do not exist.

[ENOTDIR] A component of the path prefix of the new process file is not a directory.

[EACCES] Search permission is denied for a directory listed in the new process file's path prefix.

[EACCES] The new process file is not an ordinary file.

[ENOMEM] The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.

[E2BIG] The number of bytes in the new process's argument list is greater than the system-imposed limit of 5120 bytes.

[EFAULT] The new process file is not as long as indicated by the size values in its header.

[EFAULT] *Path*, *argv*, or *envp* points to an illegal address.

RETURN VALUE

If *exec* returns to the calling process an error has occurred; the return value will be −1 and *errno* will be set to indicate the error.

SEE ALSO

sh(1).
exit(2), fork(2), signal(2), umask(2).
environ(5).

NAME

exit, _exit – terminate process

SYNOPSIS

```
void exit (status)
int status;
void _exit (status)
int status;
```

DESCRIPTION

Exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination. The low order eight bits (i.e., bits 0377) of *status* are made available to the parent process; see *wait*(2).

If the parent process of the calling process is not executing a *wait*, information about the terminated child is held until a *wait* is executed. If several processes have terminated before *wait* is called, information about one of them, chosen at random, will be returned.

The C function *exit* may cause cleanup actions before the process exits. The function *_exit* circumvents all cleanup.

SEE ALSO

intro(2), signal(2), wait(2).

WARNING

See *WARNING* in *signal*(2).

NAME

fcntl – file control

SYNOPSIS

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;

DESCRIPTION

Fcntl provides control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The commands (values for *cmd*) available are:

F_DUPFD Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write, or read/write).

Same file status flags (i.e., both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.

F_GETFD Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0** the file will remain open across *exec*; otherwise, the file will be closed upon execution of *exec*.

F_SETFD Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (**0** or **1** as above).

F_GETFL Get file status flags.

F_SETFL Set file status flags to *arg*. Only certain flags can be set; see *fcntl*(5).

Fcntl will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EMFILE] *Cmd* is F_DUPFD and 20 file descriptors are currently open.

[EMFILE] *Cmd* is F_DUPFD and *arg* is negative or greater than 20.

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_SETFD	Value other than −1.
F_GETFL	Value of file flags.
F_SETFL	Value other than −1.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), exec(2), open(2).
fcntl(5).

NAME

fork – create a new process

SYNOPSIS

int fork ()

DESCRIPTION

Fork creates a new process. The new process (child process) is an exact copy of the calling process (parent process).

The child process inherits the following attributes from the parent process:

environment
close-on-exec flag (see *exec*(2))
signal handling settings (see *signal*(2))
nice value (see *nice*(2))
process group ID
tty group ID (see *exit*(2) and *signal*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))

The child process differs from the parent process in the following ways:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

Fork will fail and no child process will be created if one or more of the following are true:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded.

[EAGAIN] The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

RETURN VALUE

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

SEE ALSO

exec(2), nice(2), signal(2), umask(2), wait(2).

NAME

getpid, getpgrp, getppid – get process, process group, and parent process IDs

SYNOPSIS

int getpid ()

int getpgrp ()

int getppid ()

DESCRIPTION

Getpid returns the process ID of the calling process.

Getpgrp returns the process group ID of the calling process.

Getppid returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), signal(2).

NAME

kill – send a signal to a process or a group of processes

SYNOPSIS

int kill (pid, sig)
int pid, sig;

DESCRIPTION

Kill sends a signal to a process or group of processes specified by *pid*. The signal to be sent is specified by *sig*, and is either 0 or a signal from the list given in *signal*(2).

If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The user ID of the sending process must match the user ID of the receiving process.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*.

If *pid* is 0, *sig* will be sent to all processes whose process group ID is equal to the process group ID of the sender.

If *pid* is negative but not −1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

Kill will fail and no signal will be sent if one or more of the following are true:

[EINVAL] *Sig* is not a valid signal number.

[ESRCH] No process can be found corresponding to that specified by *pid*.

[EPERM] The user ID of the sending process does not match the user ID of the receiving process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1).
getpid(2), signal(2).

NAME

link – link to a file

SYNOPSIS

int link (path1, path2)
char *path1, *path2;

DESCRIPTION

Link is not supported in QNIX.

SEE ALSO

unlink(2).

NAME

lseek – move read/write file pointer

SYNOPSIS

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

DESCRIPTION

Lseek sets the file pointer associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call.

Upon successful completion, the resulting pointer location, measured in bytes from the beginning of the file, is returned.

Lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] *Fildes* is not an open file descriptor.

[ESPIPE] *Fildes* is associated with a pipe or fifo.

[EINVAL] and SIGSYS signal
Whence is not 0, 1, or 2.

[EINVAL] The resulting file pointer would be negative.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

RETURN VALUE

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

NAME

mknod – make a directory or a special file

SYNOPSIS

int mknod (path, mode, dev)
char *path;
int mode, dev;

DESCRIPTION

Mknod creates a new file named by the pathname pointed to by *path*. The mode of the new file is initialized from *mode*. The value of *mode* is interpreted as follows:

0170000 File type; one of the following:
0010000 fifo special
0040000 directory
0100000 or 0000000 ordinary file
0000777 Access permissions; constructed from the following
0000400 read by owner
0000200 write by owner
0000100 execute (search on directory) by owner
0000007 read, write, execute (search) by others

The owner ID of the file is set to the user ID of the calling process.

Values of *mode* other than those above are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process's file mode creation mask; all bits set in the file mode creation mask are cleared. See *umask*(2).

Mknod will fail and the new file will not be created if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[EINVAL] The specified mode of the file is invalid.

[ENOENT] A component of the path prefix does not exist.

[EEXIST] The named file already exists.

[EFAULT] *Path* points outside the allocated address space of the process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

mkdir(1).
chmod(2), exec(2), umask(2).

NAME

open – open for reading or writing

SYNOPSIS

```
#include <fcntl.h>
int open (path, oflag [ , mode ] )
char *path;
int oflag, mode;
```

DESCRIPTION

Open opens a file descriptor for a file and sets the file status flags according to the value of *oflag*. The file is named by the pathname pointed to by *path*. *Oflag* values are constructed by or-ing together flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes. See *read*(2) and *write*(2).

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise, the owner ID of the created file is set to the user ID of the process, and the low-order 12 bits of the file mode are set to the value of *mode* modified so that all bits set in the file mode creation mask of the process are cleared (see *umask*(2) and *creat*(2)).

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

The named file is opened unless one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] O_CREAT is not set and the named file does not exist.

[EACCES] Search permission is denied on a component of the path prefix.

[EACCES] *Oflag* permission is denied for the named file.

[EISDIR] The named file is a directory and *oflag* is write or read/write.

[EMFILE] Twenty file descriptors are currently open.

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[EFAULT] *Path* points outside the allocated address space of the process.

[EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

[ENXIO] O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

[EINTR] A signal was caught during the *open* system call.

[ENFILE] The system file table is full.

RETURN VALUE

Upon successful completion, the file descriptor is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), lseek(2), read(2), umask(2), write(2).

NAME

pause – suspend process until signal

SYNOPSIS

pause ()

DESCRIPTION

Pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process (see *signal*(2)).

In QNIX, the *setpause*(2) system call can be used to set a mask to control which signals cause *pause* to resume.

RETURN VALUE

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function (see *signal*(2)), the calling process resumes execution from the point of suspension, with a return value of −1 from *pause* and *errno* set to EINTR.

SEE ALSO

kill(2), signal(2), wait(2), setpause(2).

NAME

pipe – create an interprocess channel

SYNOPSIS

int pipe (fildes)
int fildes[2];

DESCRIPTION

Pipe creates an I/O mechanism called a pipe. It returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

A read-only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis. No more than 5120 bytes of data should be written to a pipe in a single write call.

Pipe will fail if:

[EMFILE] 19 or more file descriptors are currently open.

[ENFILE] The system file table is full.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

read(2), write(2).
sh(1) in the *UNIX System User Reference Manual.*

NAME

read – read from file

SYNOPSIS

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

Read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*. *Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

On devices capable of seeking, the *read* starts at the position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer. This number may be less than *nbyte* if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

When attempting to read from an empty pipe (or FIFO):

If O_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

If O_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block until data becomes available.

See *open*(2) for an explanation of O_NDELAY. *Read* will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid file descriptor open for reading.

[EFAULT] *Buf* points outside the allocated address space.

[EINTR] A signal was caught during the *read* system call.

RETURN VALUE

Upon successful completion, a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a -1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2), pipe(2).

NAME

setpause, checkpause – control signal actions

SYNOPSIS

setpause (action, signum)
int action, signum;

checkpause(signum)
int signum;

DESCRIPTION

Setpause controls whether a given signal will wake the calling process from a suspension caused by *wait*(2), *pause*(2), or the read of a pipe. If *action* is non-zero, the signal specified by *signum* will cause the process to be restarted. If *action* is zero, the signal will be held pending to the process until it is awakened by other means.

Checkpause may be used to check the state of the pause control bit associated with the signal specified by *signum*.

SEE ALSO

signal(2), wait(2), pause(2).

NAME

signal – specify what to do upon receipt of a signal

SYNOPSIS

```
#include <signal.h>

int (*signal (sig, func))()
int sig;
void (*func)();
```

DESCRIPTION

Signal allows the calling process to choose how to handle the receipt of signals. *Sig* names a specific signal, and *func* specifies one of three possible ways to handle the signal.

Sig can be assigned any one of the following except **SIGKILL**:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03*	quit
SIGILL	04*	illegal instruction (not reset when caught)
SIGTRAP	05*	trace trap (not reset when caught)
SIGIOT	06*	IOT instruction
SIGEMT	07*	EMT instruction
SIGFPE	08*	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18	death of a child (see *WARNING* below)
SIGPWR	19	power fail (see *WARNING below)*
SIGUSR20	20	user-defined signal 3 (see *WARNING below)*
SIGUSR31	31	user-defined signal 14 (see *WARNING below)*

The signals marked by an asterisk (*) have a special characteristic explained below.

Func is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL – terminate process upon receipt of signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). In addition, if *sig* is marked with an asterisk, a debugger will be started for the current process.

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signal **SIGKILL** cannot be ignored.

function address – catch signal

Upon receipt of *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Additional arguments are passed to the signal-catching function for hardware-generated signals. Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

When a signal that is to be caught occurs during a *read*, *write*, or *open* system call on a slow device (like a terminal; but not a file), during a *pause* system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a −1 to the calling process with *errno* set to EINTR.

QNIX allows the interruption of *pause* or *wait* system calls to be controlled by a bit mask (see *pause*(2) and *setpause*(2)).

Note: The signal **SIGKILL** cannot be caught.

A call to *signal* cancels a pending signal except for a pending **SIGKILL** signal.

Signal will fail if *sig* is an illegal signal number, including **SIGKILL**. [EINVAL]

RETURN VALUE

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1).
kill(2), pause(2), setpause(2), wait(2).
setjmp(3C).

WARNING

In this release of QNIX, two signals behave differently than described above. They are:

SIGCLD	18	death of a child (reset when caught)
SIGPWR	19	power fail (not reset when caught)

The behavior of these signals is described below. There is no guarantee that they will continue to behave in this way in future releases of QNIX. They are included only for compatibility with other versions of the UNIX system. Their use in new programs is strongly discouraged.

For these signals, the values of *func* have the following meanings:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate; see *exit*(2).

function address - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as for a normal *function address*. The same is true if the signal is **SIGCLD**, except that while the process is executing the signal-catching function, any received **SIGCLD** signals will be queued and the signal-catching function will be continually reentered until the queue is empty.

SIGCLD affects two other system calls (*wait*(2), and *exit*(2)) in the following ways:

wait If the *func* value of **SIGCLD** is set to **SIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of −1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

The signals **SIGUSR20** through **SIGUSR31** are added for QNIX programs, and are not supported in other versions of UNIX.

NAME

stat, fstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

DESCRIPTION

Stat obtains information about the file named by the pathname pointed to by *path*. Read, write, or execute permission of the named file is not required, but all directories in the pathname must be searchable.

Fstat obtains information about an open file specified by the file descriptor *fildes*. *Fildes* is obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

The information obtained by *stat* or *fstat* is placed in a stat structure pointed to by *buf*. This stat structure contains the following items:

```
ushort   st_mode;    /* File mode (see mknod(2)) */
ino_t    st_ino;     /* Inode number */
dev_t    st_dev;     /* ID of device containing */
                     /* a directory entry for this file */
dev_t    st_rdev;    /* ID of device */
                     /* This entry is defined only for */
                     /* character special or block special */
                     /* files */
short    st_nlink;   /* Number of links */
ushort   st_uid;     /* User ID of the file's owner */
ushort   st_gid;     /* Group ID of the file's group */
                     /* Not supported in QNIX */
off_t    st_size;    /* File size in bytes */
time_t   st_atime;   /* Time of last access */
time_t   st_mtime;   /* Time of last data modification */
time_t   st_ctime;   /* Time of last file status change */
                     /* Times measured in seconds since */
                     /* 00:00:00 GMT, Jan. 1, 1970 */
```

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), and *read*(2).

st_mtime Time when data was last modified. Changed by the following system calls: *chmod*(2), *creat*(2), *mknod*(2), *pipe*(2), *unlink*(2), and *write*(2).

st_ctime Same as st_mtime on QNIX.

Stat will fail if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EFAULT] *Buf* or *path* points to an invalid address.

Fstat will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EFAULT] *Buf* points to an invalid address.

RETURN VALUE

Returns a value of 0 upon successful completion. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

BUGS

In QNIX, st_nlink is always set to 1 for files and 2 for directories. In standard UNIX systems, the link count for a directory is (2 + number of sub-directories).
Only st_mode is valid for directories.

SEE ALSO

chmod(2), creat(2), mknod(2), pipe(2), read(2), time(2), unlink(2), write(2).

NAME

time – get time

SYNOPSIS

long time ((long *) 0)

long time (tloc)
long *tloc;

DESCRIPTION

Time returns the number of seconds that have elapsed since 00:00:00 GMT, January 1, 1970.

If *tloc* (taken as an integer) is non-zero, the return value is also stored in the location to which *tloc* points.

Time will fail if:

[EFAULT] *Tloc* points to an illegal address.

RETURN VALUE

Upon successful completion, *time* returns the value of time. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

umask – set file mode creation mask

SYNOPSIS

int umask (cmask)
int cmask;

DESCRIPTION

Umask sets the calling process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

mkdir(1), sh(1). chmod(2), creat(2), mknod(2), open(2).

NAME

uname – get name of current UNIX system

SYNOPSIS

```
#include <sys/utsname.h>

int uname (name)
struct utsname *name;
```

DESCRIPTION

Uname stores information identifying the current UNIX system in the structure pointed to by *name*.

Uname uses the structure defined in **<sys/utsname.h>**. This structure's members are:

```
char    sysname[9];
char    nodename[9];
char    release[9];
char    version[9];
char    machine[9];
```

Sysname contains a null-terminated character string naming the current UNIX system. *Nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system. *Machine* contains a standard name identifying the hardware that the UNIX system is running on.

Uname will fail if:

[EFAULT] *Name* points to an invalid address.

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, −1 is returned and *errno* is set to indicate the error.

NAME

unlink – remove directory entry

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

Unlink removes the directory entry named by the pathname pointed to by *path*.

The named file is deleted from the Accent file system. If the file has users in the QNIX environment, they may continue to use the file until the last user is finished. Subsequent attempts to open the file will fail unless a new file with the same name has been created.

Unlink will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[ETXTBSY]	The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.
[EPERM]	The named file is a directory that is not empty.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

rm(1).
close(2), link(2), open(2).

NAME

wait – wait for child process to stop or terminate

SYNOPSIS

```
int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)
```

DESCRIPTION

Wait suspends the calling process until one of its immediate children terminates, or until a child that is being traced stops because it has hit a break point. *Wait* will return prematurely if a signal is received.

If a child process stopped or terminated prior to the call to *wait*, it returns immediately.

If *stat_loc* (taken as an integer) is non-zero, 16 bits of information called *status* are stored in the low order 16 bits of the location pointed to by *stat_loc*. *Status* can be used to differentiate between stopped and terminated child processes, and also to identify the cause of termination and pass useful information to the parent. This is accomplished in the following manner:

If the child process stopped, the high order 8 bits of status contain the number of the signal that caused the process to stop. The low order 8 bits are set to 0177.

If the child process terminated due to an *exit* call, the low order 8 bits of status are zero. The high order 8 bits contain the low order 8 bits of the argument that the child process passed to *exit* (see *exit*(2)).

If the child process terminated due to a signal, the high order 8 bits of status are zero. The low order 8 bits contain the number of the signal that caused the termination (see *signal*(2)).

In QNIX, the *setpause* system call can be used to set a mask to control which signals cause *wait* to return prematurely.

Wait will fail and return immediately if one or more of the following are true:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EFAULT] *Stat_loc* points to an illegal address.

RETURN VALUE

If *wait* returns due to the receipt of an enabled signal, a value of −1 is returned to the calling process and *errno* is set to EINTR. Enabling and disabling of signals is achieved by the *setpause* system call; see *setpause*(2). If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), pause(2), setpause(2), signal(2).

WARNING

See *WARNING* in *signal*(2).

NAME

write – write on a file

SYNOPSIS

int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

Write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with *fildes*. *Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

On devices capable of seeking, the data is written at the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written. On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND file status flag is set, the file pointer will be set to the end of the file prior to each write (see *open*(2)).

Write will fail and the file pointer remain unchanged if one or more of the following are true:

[EBADF] *Fildes* is not a valid file descriptor open for writing.

[EPIPE] or SIGPIPE signal
An attempt is made to write to a pipe that is not open for reading by any process.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

[EFAULT] *Buf* points outside the process's allocated address space.

[EINTR] A signal was caught during the *write* system call.

If a *write* requests that more bytes be written than there is room for (e.g., due to the physical end of a medium), only as many bytes as there is room for will be written. For example, if there is space for only 20 bytes more in a file before reaching a limit, an attempt to write 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then a write to a full pipe (or FIFO) will return a

count of 0. If O_NDELAY is clear, writes to a full pipe (or FIFO) will block until space becomes available. See *open*(2).

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), lseek(2), open(2), pipe(2).

NAME

intro − introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

This section describes functions found in various libraries other than those functions that directly invoke UNIX system primitives. Functions that directly invoke UNIX system primitives are described in Section 2 of this manual. Certain major collections of functions are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, ~~which is automatically loaded by the C compiler, *cc*(1)~~. The link editor *ld*(1) searches this library under the **−lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(3S) These functions constitute the "standard I/O package" (see *stdio*(3S)). They are also in the library *libc*. Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.

DEFINITIONS

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as '\0'. A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A **NULL** pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. **NULL** is defined as **0** in **<stdio.h>**; the user can include an appropriate definition if not using **<stdio.h>**.

FILES

/lib/libc.a

~~/lib/libm.a~~

SEE ALSO

cc(1), ld(1), lint(1).
intro(2), stdio(3S).

cpp

pcc

.s file

as

lnk

intro(3X).

DIAGNOSTICS

Functions in the C Library (3C) may return the conventional values **0** or **±HUGE** (the largest-magnitude single-precision floating-point numbers; **HUGE** is defined in the *<math.h>* header file) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* (see *intro*(2)) is set to the value EDOM or ERANGE.

WARNING

Many of the functions in the libraries call and/or refer to other functions and external variables described in this section and in Section 2. If a program inadvertantly defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded. The *lint*(1) program checker reports name conflicts of this kind as "multiple declarations" of the names in question. Definitions for sections 2, 3C, and 3S are checked automatically. Other definitions can be included by using the −l option to *lint*. Use of *lint* is highly recommended.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – classify characters

SYNOPSIS

```
#include <ctype.h>

int isalpha (c)
int c;

. . .
```

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value **EOF** (**−1** – see *stdio*(3S)).

Macro:	True if:
isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, newline, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) through 0176 (tilde).
isgraph	*c* is a printing character, like *isprint*, except false for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).
isascii	*c* is an ASCII character, code less than 0200.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO

stdio(3S).
ascii(5).

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

DESCRIPTION

Fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed. *Fclose* is performed automatically for all open files upon calling *exit*(2).

Fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

DIAGNOSTICS

These functions return 0 for success, and **EOF** if any error (such as trying to write to a file that has not been opened for writing) was detected.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S).

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int ferror (stream)
FILE *stream;

int feof (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

int fileno (stream)
FILE *stream;
```

DESCRIPTION

Ferror returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*; otherwise it returns zero.

Feof returns non-zero when **EOF** has previously been detected reading the named input *stream*; otherwise it returns zero.

Clearerr resets the error indicator and **EOF** indicator to zero on the named *stream*.

Fileno returns the integer file descriptor associated with the named *stream*; see *open*(2).

NOTE

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open(2).
fopen(3S).

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen (file-name, type)
char *file-name, *type;

FILE *freopen (file-name, type, stream)
char *file-name, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

DESCRIPTION

Fopen opens the file named by *file-name* and associates a *stream* with it. *Fopen* returns a pointer to the FILE structure associated with *stream*.

File-name points to a character string that contains the name of the file to be opened.

Type is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

Freopen substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open succeeds. *Freopen* returns a pointer to the FILE structure associated with *stream*. *Freopen* is typically used to attach the preopened streams associated with **stdin**, **stdout** and **stderr** to other files.

Fdopen associates a *stream* with a file descriptor. File descriptors are obtained from *open*, *dup*, *creat*, or *pipe*(2), which open files but do not return pointers to a FILE structure stream. Streams are necessary input for many of the Section 3S library routines. The *type* of *stream* must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation that encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *Fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and the file pointer is repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

creat(2), dup(2), open(2), pipe(2).
fclose(3S), fseek(3S).

DIAGNOSTICS

Fopen and *freopen* return a NULL pointer on failure.

NAME

fread, fwrite – binary input/output

SYNOPSIS

```
#include <stdio.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

DESCRIPTION

Fread copies *nitems* items of data from the named input *stream* into an array pointed to by *ptr*. An item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. *Fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. *Fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. *Fread* does not change the contents of *stream*.

Fwrite appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. *Fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. *Fwrite* does not change the contents of the array pointed to by *ptr*.

The argument *size* is typically *sizeof(*ptr)*, where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than *char*, it should be cast into a pointer to *char*.

SEE ALSO

read(2), write(2).
fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S).

DIAGNOSTICS

Fread and *fwrite* return the number of items read or written. If *size* or *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

NAME

fseek, rewind, ftell – reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according to *ptrname* (the values 0, 1, or 2, respectively).

Rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0), except that no value is returned.

Fseek and *rewind* undo any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

Ftell returns the offset relative to the beginning of the file of the current byte associated with the named *stream*.

SEE ALSO

lseek(2).
fopen(3S), popen(3S), ungetc(3S).

DIAGNOSTICS

Fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*. In particular, *fseek* may not be used on a terminal, or on a file opened via *popen*(3S).

WARNING

Although on the UNIX system an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX systems requires that an offset be used by *fseek*

directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

getc, getchar, fgetc, getw – get character or word from a stream

SYNOPSIS

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next character (i.e., byte) from the named input *stream* as an integer. It also moves the file pointer, if defined, ahead one character in *stream.*

Getchar is defined as *getc(stdin)*. *Getc* and *getchar* are macros.

Fgetc behaves like *getc*, but is a function rather than a macro. *Fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

Getw returns the next word (i.e., integer) from the named input *stream.* *Getw* increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. QNIX integers are 32 bit. *Getw* assumes no special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S).

DIAGNOSTICS

These functions return the constant **EOF** at end-of-file or upon an error. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

WARNING

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed because sign-extension of a character on widening to integer is machine-dependent.

BUGS

Because it is implemented as a macro, *getc* treats a *stream* argument with side effects incorrectly. In particular, **getc(*f++)** does not work sensibly. *Fgetc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

getenv – return value for environment name

SYNOPSIS

```
char *getenv (name)
char *name;
```

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *name*=*value*, and returns a pointer to the *value* in the current environment. If such a string is not present, *genenv* returns a NULL pointer.

SEE ALSO

exec(2).
environ(5).

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, *stdin*, into the array pointed to by *s*, until either a newline character is read or an end-of-file condition is encountered. The newline character is discarded, and in either case the string is terminated with a null character.

Fgets reads characters from the *stream* into the array pointed to by *s*, until either $n-1$ characters are read, a newline character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

```
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package.

Malloc returns a pointer to a block of at least *size* bytes suitably aligned for any use. It allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* (see *brk*(2)) to get more memory from the system when there is no suitable space already free.

Free releases a block previously allocated by *malloc* for reallocation, but does not disturb its contents. The *ptr* argument is a pointer to the block to be released.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents of the block will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, *realloc* will ask *malloc* to enlarge the arena by *size* bytes and will then move the data to the new space. *Realloc* also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*. Thus, sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

SEE ALSO

brk(2).

DIAGNOSTICS

Malloc, *realloc* and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

NOTE

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer.

NAME

memccpy, memchr, memcmp, memcpy, memset – memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memccpy copies characters from memory area **s2** into **s1**, stopping after the first occurrence of character **c** has been copied, or after **n** characters have been copied, whichever comes first. It returns a pointer to the character after the copy of **c** in **s1**, or a NULL pointer if **c** was not found in the first **n** characters of **s2**.

Memchr returns a pointer to the first occurrence of character **c** in the first **n** characters of memory area **s**, or a NULL pointer if **c** does not occur in the first **n** characters.

Memcmp compares its arguments, looking at the first **n** characters only, and returns an integer less than, equal to, or greater than 0, depending on whether **s1** is lexicographically less than, equal to, or greater than **s2**, respectively.

Memcpy copies **n** characters from memory area **s2** to **s1**. It returns **s1**.

Memset sets the first **n** characters in memory area **s** to the value of character **c**. It returns **s**.

NOTE

For convenience, all these functions are declared in the optional *<memory.h>* header file.

BUGS

Memcmp uses native character comparison, which is performed by Accent microcode. Native character comparison is signed in QNIX and on PDP-11s and VAX-11s, and is unsigned on other machines. Thus, the sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.
Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.
Characters sign extend when copied into longer data items.

NAME

mktemp – make a unique filename

SYNOPSIS

char *mktemp (template)
char *template;

DESCRIPTION

Mktemp replaces the contents of the string pointed to by *template* by a unique filename, and returns the address of *template*. The string in *template* should be a filename with six trailing Xs; *mktemp* replaces the Xs with a letter and the current process ID. The letter is chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2).

BUGS

It is possible to run out of letters.

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

```
void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;
```

DESCRIPTION

Perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a newline. To be of most use, *s* should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string. *Sys_nerr* is the largest message number provided for in *sys_errlist*; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2).

NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf (format [ , arg ] ... )
char *format;

int fprintf (stream, format [ , arg ] ... )
FILE *stream;
char *format;

int sprintf (s, format [ , arg ] ... )
char *s, format;
```

DESCRIPTION

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output," followed by the null character (\0), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (excluding the \0 in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *arg*s under control of *format*. *Format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in the fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '–', described below, has been given) to the field width. If the field width for an s conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.

A *precision* that gives the minimum number of digits to appear for the **d**, **o**, **u**, **x**, or **X** conversions, the number of digits to appear

after the decimal point for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero.

An optional **l** specifying that a following **d**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*. An **l** before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

− The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (+ or −).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. If the blank and + flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** or **X** conversion, a non-zero result will have **0x** or **0X** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (they normally are removed).

The conversion characters and their meanings are:

d,o,u,x,X The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be

represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

f The float or double *arg* is converted to decimal notation in the style "[−]ddd.ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.

e,E The float or double *arg* is converted in the style "[−]d.ddde±dd," where there is one digit before the decimal point and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than −4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

c The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

% Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02," where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

SEE ALSO

putc(3S), scanf(3S), stdio(3S).

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc (c, stream)
int c;
FILE *stream;

int putchar (c)
int c;

int fputc (c, stream)
int c;
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Putc writes the character *c* onto the output *stream* (at the position at which the file pointer, if defined, is pointing). *Putchar*(*c*) is defined as *putc*(*c, stdout*). *Putc* and *putchar* are macros.

Fputc behaves like *putc*, but is a function rather than a macro. *Fputc* runs more slowly than *putc*, but it takes less space per invocation and its name can be passed as an argument to a function.

Putw writes the word (i.e., integer) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. QNIX integers are 32 bit. *Putw* neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen* (see *fopen*(3S)) will cause it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a newline character is written or terminal input is requested). *Setbuf*(3S) may be used to change the stream's buffering

strategy.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant **EOF**. Failure will occur if the file *stream* is not open for writing or if the output file cannot be grown. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

BUGS

Because it is implemented as a macro, *putc* will treat a *stream* argument with side effects incorrectly. In particular, **putc(c, *f++);** does not work correctly. *Fputc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Puts writes the null-terminated string pointed to by *s*, followed by a newline character, to the standard output stream *stdout*.

Fputs writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

DIAGNOSTICS

Both routines return **EOF** on error. This will happen if the routines try to write on a file that has not been opened for writing.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

NOTES

Puts appends a newline character while *fputs* does not.

NAME

rand, srand – simple random-number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned seed;

DESCRIPTION

Rand uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to $2^{15}-1$.

Srand can be called at any time to reset the random-number generator to a new starting point. The generator is initially seeded with a value of 1.

NOTE

The spectral properties of *rand* leave a great deal to be desired.

NAME

scanf, fscanf, sscanf – convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf (format [ , pointer ] ... )
char *format;

int fscanf (stream, format [ , pointer ] ... )
FILE *stream;
char *format;

int sscanf (s, format [ , pointer ] ... )
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string *format* usually contains conversion specifications that are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, newlines, or form-feeds), which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional l or h indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. Suppressing assignment provides a way of skipping an input field. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all conversions except "[" and "c", white space leading an input field is

ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

% a single **%** is expected in the input at this point; no assignment is done.

d a decimal integer is expected; this requires an integer pointer argument.

u an unsigned decimal integer is expected; this requires an unsigned integer pointer argument.

o an octal integer is expected; this requires an integer pointer argument.

x a hexadecimal integer is expected; this requires an integer pointer argument.

e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional **+**, **−**, or space, followed by an integer.

s a character string is expected; this requires a character pointer argument pointing to an array of characters large enough to accept the string and a terminating **\0**, which is added automatically. The input field is terminated by a white-space character.

c a character is expected; this requires a character pointer argument. This conversion does not skip over white space; to read the next non-space character, use **%1s**. If a field width is given, the argument should be a pointer to a character array; the indicated number of characters is read.

[indicates string data. Leading white space is not skipped. The left bracket is followed by a set of characters, called the *scanset*, and a right bracket; the input field is the maximum sequence of input characters consisting entirely of characters in the scanset. A circumflex (**^**) as the first character in the scanset redefines the scanset as the set of all characters *not* in the scanset string. In the scanset, a range of characters may be represented by *first*−*last*; thus [0123456789] may be expressed [0−9]. *First* must be lexically less than or equal to *last*, or else the dash will stand for

itself. The dash will also stand for itself whenever it is the first or last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset. This conversion requires an argument pointer to a character array large enough to hold the data field and the terminating **\0**, which is added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, and **x** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short**, rather than a pointer to **int**, is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than a pointer to **float** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

Scanf conversion terminates at **EOF**, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the conflicting character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign the value **3** to *n*, the value **25** to *i*, the value **5.432** to *x*, and the value **thompson\0** to *name*. Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0-9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* (see *getc*(3S)) will return **a**.

SEE ALSO

getc(3S), printf(3S), strtod(3C), strtol(3C).

NOTE

Trailing white space (including a newline) is left unread unless matched in the control string.

DIAGNOSTICS

These functions return **EOF** on end of input and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf, setvbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;
```

DESCRIPTION

Setbuf causes the array pointed to by *buf* to be used for buffering *stream*, instead of an automatically allocated buffer. *Setbuf* may be used after a stream has been opened but before it is read or written. If *buf* is the NULL pointer, input/output will be completely unbuffered.

A constant **BUFSIZ**, defined in the **<stdio.h>** header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

Setvbuf may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type* (defined in stdio.h) are:

_IOFBF	causes input/output to be fully buffered.
_IOLBF	causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.
_IONBF	causes input/output to be completely unbuffered.

If *buf* is not the **NULL** pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *Size* specifies the size of the buffer to be used. The constant **BUFSIZ** in **<stdio.h>** is suggested as a good buffer size. If input/output is to be unbuffered, *buf* and *size* are ignored.

By default, output to a terminal is line buffered and all other input/output is fully buffered.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S), stdio(3S).

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

NOTE

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* (whose type, *jmp_buf*, is defined in the *<setjmp.h>* header file) for later use by *longjmp*. It returns the value 0.

Longjmp restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. All accessible data had values as of the time *longjmp* was called.

SEE ALSO

signal(2).

WARNING

If *longjmp* is called without *setjmp* having placed anything in *env*, or when the last *setjmp* call was in a function which has since returned, unpredictable things will happen.

NAME

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use or act as if they use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

The constant **NULL** (0) designates a nonexistent pointer.

An integer constant **EOF** (−1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

An integer constant **BUFSIZ** specifies the size of the buffers used by the particular implementation.

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that header file and need no further declaration. The constants and the following "functions" are implemented as macros: *getc*, *getchar*, *putc*, *putchar*, *ferror*, *feof*, *clearerr*, and *fileno*. Redeclaration of these names is perilous.

SEE ALSO

open(2), close(2), lseek(2), pipe(2), read(2), write(2).
fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), ungetc(3S).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok – string operations

SYNOPSIS

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

char *strrchr (s, c)
char *s;
int c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;
```

char *strtok (s1, s2)
char *s1, *s2;

DESCRIPTION

The arguments **s1**, **s2** and **s** in these functions point to strings (arrays of characters terminated by a null character).

The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**.

Strcat appends a copy of string **s2** to the end of string **s1**. *Strncat* appends at most **n** characters. Each returns a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, depending on whether **s1** is lexicographically less than, equal to, or greater than **s2**, respectively. *Strncmp* makes the same comparison but examines at most **n** characters.

Strcpy copies string **s2** to **s1**, stopping after the null character has been copied. *Strncpy* copies exactly **n** characters, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null-terminated if the length of **s2** is **n** or more. Each function returns **s1**.

Strlen returns the number of characters in **s**, not including the terminating null character.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character **c** in string **s**, or a NULL pointer if **c** does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a NULL pointer if no character from **s2** exists in **s1**.

Strspn (*strcspn*) returns the length of the initial segment of string **s1** that consists entirely of characters from (not from) string **s2**.

Strtok considers the string **s1** to be a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2**. The first call, with a pointer to **s1** specified as the first argument, returns a pointer to the first character of the first token, and writes a null character into **s1** immediately after the first token. *Strtok* keeps track of its position in the string **s1**. Subsequent calls to *strtok* (made with a NULL pointer as the first argument, in place of a pointer to a string) will work through the original string **s1** until no tokens remain. The separator string **s2** may be changed from call to call. When no tokens remain in **s1**, a NULL pointer is returned. When a new string is given as the first argument to *strtok*, its

position in a past string is completely forgotten.

NOTE

For user convenience, all these functions are declared in the optional *<string.h>* header file.

BUGS

Strcmp and *strncmp* use native character comparison, which is signed in QNIX and on PDP-11s and VAX-11s, unsigned on other machines. Thus the sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

```
long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;
```

DESCRIPTION

Strtol returns as a long integer the value represented by the character string pointed to by *str*. Leading white-space characters (as defined by *isspace* in *ctype*(3C)) are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base as follows: after an optional leading sign, a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment or by an explicit cast.

Atol(str) is equivalent to *strtol(str, (char **)NULL, 10).*

Atoi(str) is equivalent to *(int) strtol(str, (char **)NULL, 10).*

SEE ALSO

ctype(3C), scanf(3S).

BUGS

Overflow conditions are ignored.

NAME

swab – swap bytes

SYNOPSIS

```
void swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP-11s and other machines.

Nbytes should be even and non-negative. If *nbytes* is odd and positive, *swab* uses *nbytes*−1 instead. If *nbytes* is negative, *swab* does nothing.

NAME

ungetc – push character back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;
```

DESCRIPTION

Ungetc inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc*(3S) call on the *stream*. *Ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered. If *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

If *c* equals **EOF**, *ungetc* does nothing to the buffer and returns **EOF**.

Fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S).

DIAGNOSTICS

Ungetc returns **EOF** if it cannot insert the character.

NAME

vprintf, vfprintf, vsprintf – print formatted output of a varargs argument list

SYNOPSIS

```
#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

Vprintf, *vfprintf*, and *vsprintf* are the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(5).

EXAMPLE

The following demonstrates how *vfprintf* could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *      error should be called like
 *              error(function_name, format, arg1, arg2...);
 */
/*VARARGS0*/
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 *      separately declared because of the definition of varargs.
 */
```

```
va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);
        /* print out name of function causing error */
        (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
        fmt = va_arg(args, char *);
        /* print out remainder of message */
        (void)vfprintf(fmt, args);
        va_end(args);
        (void)abort( );
}
```

SEE ALSO

vprintf(3X).
varargs(5).

NAME

intro – introduction to specialized routines and libraries

DESCRIPTION

This section describes routines in various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

SEE ALSO

intro(2), intro(3).
cc(1), ld(1), lint(1) in the *UNIX System User Reference Manual.*

WARNING

See the *Warning* in *intro*(3).

NAME

vprintf, vfprintf, vsprintf – print formatted output of a varargs argument list

SYNOPSIS

```
#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

Vprintf, *vfprintf*, and *vsprintf* are the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(5).

EXAMPLE

The following demonstrates how *vfprintf* could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *      error should be called like
 *              error(function_name, format, arg1, arg2...);
 */
/*VARARGS0*/
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 *      separately declared because of the definition of varargs.
 */
```

```
va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);
        /* print out name of function causing error */
        (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
        fmt = va_arg(args, char *);
        /* print out remainder of message */
        (void)vfprintf(fmt, args);
        va_end(args);
        (void)abort( );
}
```

SEE ALSO

printf(3S).
varargs(5).

NAME

intro – introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys.**

NAME

dir – format of directories

SYNOPSIS

#include <sys/dir.h>

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry. The structure of a directory entry as given in the include file is:

```
#ifndef DIRSIZ
#define DIRSIZ      25
#endif
struct   direct
{
        ino_t  d_ino;
        char   d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for . and .. The first is an entry for the directory itself. The second is for the parent directory. The meaning of .. is modified for the root directory of the master file system; there is no parent, so .. has the same meaning as ..

NAME

intro – introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

NAME

environ – user environment

DESCRIPTION

An array of strings called the "environment" is made available by *exec*(2) when a process begins. By convention, these strings have the form "name=value". The following names are used by various commands:

PATH The sequence of directory prefixes that *sh*(1), etc., apply in searching for a file specified by an incomplete pathname. The prefixes are separated by colons (:). **PATH** is set initially to **:/bin:/usr/bin**. Since the elements of **PATH** are separated by colons, it is not possible to include Accent searchlists in the path list.

HOME Name of the user's login directory.

Further names may be placed in the environment by the *export* command and "name=value" arguments in *sh*(1), or by *exec*(2). It is unwise to conflict with certain shell variables that are frequently exported by **.profile** files: **MAIL, PS1, PS2, IFS.**

The QNIX environment is supported by the Accent Environment Manager, and all its facilities may be exploited by direct calls. Exported QNIX environment variables are stored as local strings in the Accent Environment Manager.

SEE ALSO

exec(2).
sh(1) in the *UNIX System User Reference Manual.*

NAME

fcntl – file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR   2
#define O_NDELAY 04     /* Non-blocking I/O */
#define O_APPEND 010    /* append (writes guaranteed at the end) */

/* Flag values accessible only to open(2) */
#define O_CREAT  00400  /* open with file create (uses third open arg)*/
#define O_TRUNC  01000  /* open with truncation */
#define O_EXCL   02000  /* exclusive open */

/* fcntl(2) requests */
#define F_DUPFD  0      /* Duplicate fildes */
#define F_GETFD  1      /* Get fildes flags */
#define F_SETFD  2      /* Set fildes flags */
#define F_GETFL  3      /* Get file flags */
#define F_SETFL  4      /* Set file flags */
```

SEE ALSO

fcntl(2), open(2).

NAME

stat – data returned by stat system call

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
```

DESCRIPTION

The system calls *stat* and *fstat* return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

```
/*
 * Structure of the result of stat
 */

struct  stat
{
        dev_t   st_dev;
        ino_t   st_ino;
        ushort  st_mode;
        short   st_nlink;
        ushort  st_uid;
        ushort  st_gid;
        dev_t   st_rdev;
        off_t   st_size;
        time_t  st_atime;
        time_t  st_mtime;
        time_t  st_ctime;
};

#define S_IFMT   0170000 /* type of file */
#define S_IFDIR  0040000 /* directory */
#define S_IFCHR  0020000 /* character special */
#define S_IFBLK  0060000 /* block special */
#define S_IFREG  0100000 /* regular */
#define S_IFIFO  0010000 /* fifo */
#define S_ISUID  04000   /* set user id on execution */
#define S_ISGID  02000   /* set group id on execution */
#define S_ISVTX  01000   /* save swapped text even after use */
#define S_IREAD  00400   /* read permission, owner */
#define S_IWRITE 00200   /* write permission, owner */
```

```
#define S_IEXEC   00100    /* execute/search permission, owner */
```

FILES

/usr/include/sys/types.h
/usr/include/sys/stat.h

SEE ALSO

stat(2), types(5).

NAME

types – primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in UNIX system code. Some data of these types are accessible to user code:

```
typedef  struct { int r[1]; } *        physadr;
typedef  long                  daddr_t;
typedef  short *               caddr_t;
typedef  unsigned int          uint;
typedef  unsigned short        ushort;
typedef  ushort                ino_t;
typedef  short                 cnt_t;
typedef  long                  time_t;
typedef  int                   label_t[8];
typedef  short                 dev_t;
typedef  long                  off_t;
typedef  long                  paddr_t;
typedef  long                  key_t;
```

The form *daddr_t* is used for disk addresses except in an i-node on disk; see *fs*(4). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

SEE ALSO

fs(4).

NAME

values – machine-dependent values

SYNOPSIS

#include <values.h>

DESCRIPTION

This file contains a set of manifest constants, conditionally defined for particular processor architectures.

The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit.

BITS(*type*) The number of bits in a specified type (e.g., int).

HIBITS The value of a short integer with only the high-order bit set (in most implementations, 0x8000).

HIBITL The value of a long integer with only the high-order bit set (in most implementations, 0x80000000).

HIBITI The value of a regular integer with only the high-order bit set (usually the same as HIBITS or HIBITL).

MAXSHORT The maximum value of a signed short integer (in most implementations, 0x7FFF ≡ 32767).

MAXLONG The maximum value of a signed long integer (in most implementations, 0x7FFFFFFF ≡ 2147483647).

MAXINT The maximum value of a signed regular integer (usually the same as MAXSHORT or MAXLONG).

MAXFLOAT, LN_MAXFLOAT The maximum value of a single-precision floating-point number, and its natural logarithm.

MAXDOUBLE, LN_MAXDOUBLE The maximum value of a double-precision floating-point number, and its natural logarithm.

MINFLOAT, LN_MINFLOAT The minimum positive value of a single-precision floating-point number, and its natural logarithm.

MINDOUBLE, LN_MINDOUBLE — The minimum positive value of a double-precision floating-point number, and its natural logarithm.

FSIGNIF — The number of significant bits in the mantissa of a single-precision floating-point number.

DSIGNIF — The number of significant bits in the mantissa of a double-precision floating-point number.

FILES

/usr/include/values.h

SEE ALSO

intro(3), math(5).

NAME

varargs – handle variable argument list

SYNOPSIS

```
#include <varargs.h>

va_alist

va_dcl

void va_start(pvar)
va_list pvar;

type va_arg(pvar, type)
va_list pvar;

void va_end(pvar)
va_list pvar;
```

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists (such as *printf*(3S)) but do not use *varargs* are inherently nonportable, as different machines use different argument-passing conventions.

va_alist is used as the parameter list in a function header.

va_dcl is a declaration for *va_alist*. No semicolon should follow *va_dcl*.

va_list is a type defined for the variable used to traverse the list.

va_start is called to initialize *pvar* to the beginning of the list.

va_arg will return the next argument in the list pointed to by *pvar*. *Type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

va_end is used to clean up.

Multiple traversals, each bracketed by *va_start* ... *va_end*, are possible.

EXAMPLE

This example is a possible implementation of *execl*(2).

```
#include <varargs.h>
#define MAXARGS    100

/*      execl is called by
                execl(file, arg1, arg2, ..., (char *)0);
```

```
*/
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[MAXARGS];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while ((args[argno++] = va_arg(ap, char *)) != (char *)0)
                ;
        va_end(ap);
        return execv(file, args);
}
```

SEE ALSO

exec(2), printf(3S).

BUGS

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this from the stack frame. For example, *execl* is passed a zero pointer to signal the end of the list, and *printf* can tell how many arguments there are by the format.

It is non-portable to specify a second argument of *char*, *short*, or *float* to *va_arg*, since arguments seen by the called function are not *char*, *short*, or *float*. C converts *char* and *short* arguments to *int* and converts *float* arguments to *double* before passing them to a function.